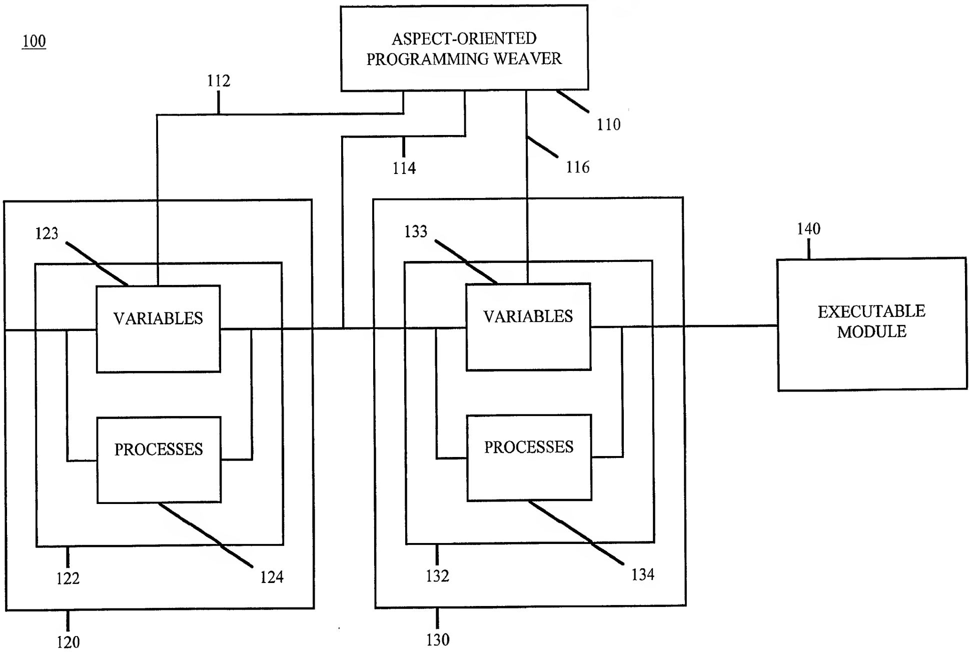
100
ASPECT-ORIENTED PROGRAMMING WEAVER
112
110
114
116
123
133
140
VARIABLES
VARIABLES
EXECUTABLE MODULE
PROCESSES
PROCESSES
122
124
132
134
120
130

FIG. 1

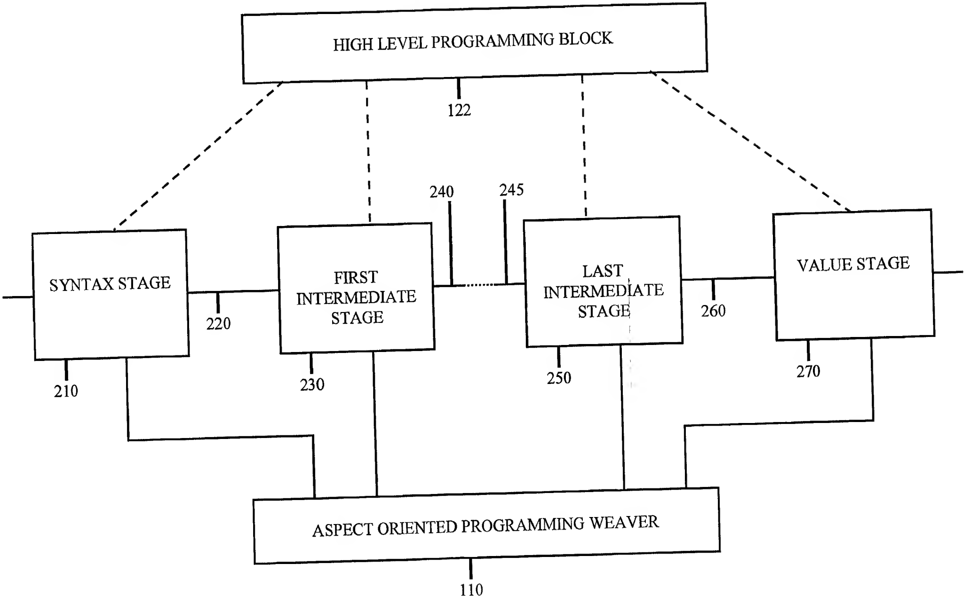
HIGH LEVEL PROGRAMMING BLOCK
122
240
245
SYNTAX STAGE
220
FIRST INTERMEDIATE STAGE
LAST INTERMEDIATE STAGE
260
VALUE STAGE
210
230
250
270
ASPECT ORIENTED PROGRAMMING WEAVER
110

FIG. 2

```
(stage requested-loops)
(projection key :defined-on requested-loops)
(projection computing-loop :defined-on requested-loops)
(define test
  (lambda (x y z)
    (and! (and! X y) z)))
(define and!
  (reduction-stage requested-loops
    (lambda (arg1 arg2)
      (pointwise #'and arg1 arg2))))
(propagator requested-loops :bottom-up
  (lambda (term)
    (case requested-loops term
      ((pointwise op arg1 arg2) (op arg1 arg2)
       (let ((starting-loop
              (fuse-loops (get-or-make-loop arg1) (get-or-make-loop arg2))
             (my-key (gensym)))
            (deconstruct requested-loops starting-loop
              (ptw-loop fn inputs outputs) (fn inputs outputs)
              (let* ((new-fn (reduction-stage computation
                                (lambda (args)
                                  (let* ((temp (fn args))
                                         (result
                                           (op (find (key arg1) temp)
                                               (find (key arg2) temp))))
                                       (cons (cons my-key result)
                                             temp))))
                   (new-loop (defer (ptw-loop new-fn inputs outputs))))
                 (update (key value) my-key)
                 (update (computing-loop value) new-loop))
                 (if (computing-loop arg1)
                   (update (computing-loop arg1)
                           (defer (loop-reference value))))
                 (if (computing-loop arg2)
                   (update (computing-loop arg2)
                           (defer (loop-reference value)))))))
     (else (note-demands value)
      )))
```

FIG. 3A

```
(define get-or-make-loop (value)
  (if (and (same-frequency value) (computing-loop value))
     (get-loop value)
     (defer (ptw-loop
            (reduction-stage computation
              (lambda (args) args)
            (list (cons (key value) value))
            nil))))

(define get-loop
  (reduction-stage computation
   (lambda (value)
     (computing-loop (get-loop-location value))))

(define get-loop-location
  (reduction-stage computation
   (lambda (value)
     (case requested-loops (computing-loop value)
       ((loop-reference next) (next)
        (get-loop-location next))
       (else value))))

(define note-demands (value)
  (case requested-loops value
     ((fn . args) (fn args)
      (record-demand fn)
      (map args #record-demand))
     ((case stage value (pattern vars body) (else otherwise))
      (stage value pattern vars body otherwise)
      (record-demand value)
      (record-demand body)
      (record-demand otherwise))
     ((lambda vars body) (vars body)
      (record-demand body))))

(define record-demand (value)
  (if (computing-loop value)
     (let ((place (get-loop-location value))
          (key (key value)))
       (case requested-loops (computing-loop place)
         ((ptw fn inputs outputs) (fn inputs outputs)
          (if (not (member key outputs))
             (let ((new-outputs (cons key outputs)))
               (update (computing-loop place)
                 (delay (ptw fn inputs new-outputs)))))))))
```

FIG. 3B

```
(define ptw-loop
  (lambda (fn inputs outputs)
    (let ((output-pairs (early-mapcar (reduction-stage computation
                           (lambda (key) (cons key (new-array)))
                         outputs))))
      (dotimes ((i 0 99))
        (let* ((input-scalars
            (early-mapcar (reduction-stage computation
                       (lambda (pair)
                         (let ((key (first pair))
                             (array (second pair)))
                           (cons key (elt array i)
                      inputs))
             (output-scalars (fn input-scalars)))
           (early-map (reduction-stage computation
                   (lambda (pair)
                    (let ((key (first pair))
                        (array (second pair)))
                      (setf (elt array i)
                          (find key output-scalars))))
                 output-pairs))))))
(define pointwise (fn op1 op2 => result)
  (reduction-stage computation ;; inlineable after loop fusion
    (find (key result) (get-loop result))))
(define fuse-loops
  (lambda (loop1 loop2)
    (if (stage-eq requested-loops loop1 loop2)
       loop1
      (deconstruct loop-structure loop1
        ((ptw-loop fn1 inputss1 outputs1) (fn1 inputs1 outputs1)
         (deconstruct loop-structure loop2
           ((ptw-loop fn2 inputs2 outputs2) (fn2 inputs2 outputs2)
            (let ((inputs (merge inputs1 inputs2))
                (outputs (append outputs1 outputs2)))
              (ptw-loop
                (preserves computation
                 (lambda (inputs) (merge (fn1 inputs) (fn2 inputs)))
                inputs outputs)))))))))
(define find
  (reduction-stage computation ;; inlineable after loop fusion
    (lambda (id list)
       (deconstruct computation list
          (cons (cons key value) rest) (key value rest)
          (if (stage-eq computation key id)
            value
            (find id rest))))))
(define merge
  ... like find
```

FIG. 3C

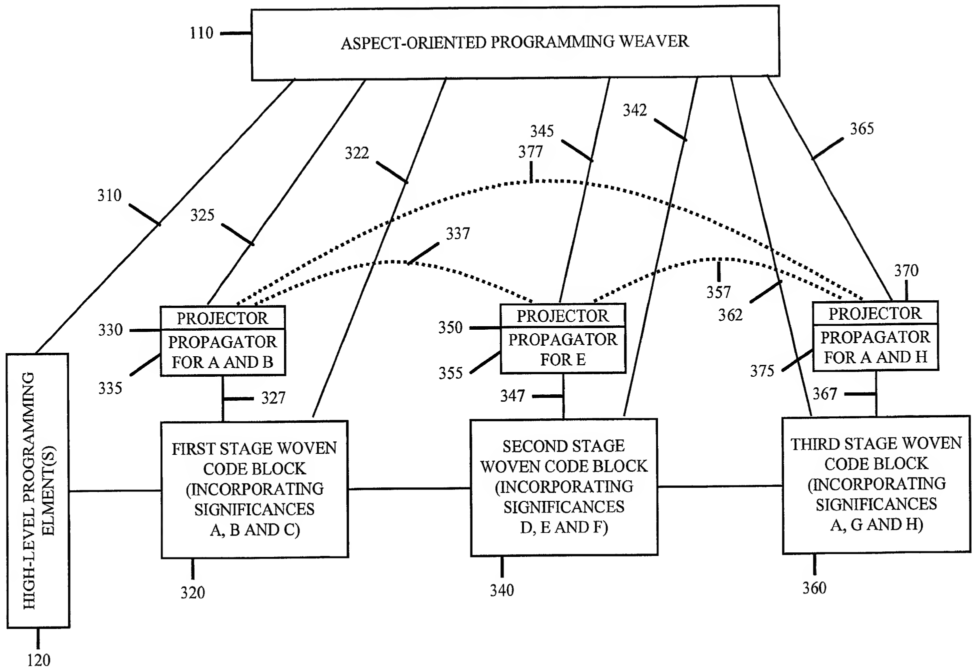
110
ASPECT-ORIENTED PROGRAMMING WEAVER
310
325
322
337
345
377
342
365
370
357
362
330
PROJECTOR
PROPAGATOR FOR A AND B
335
327
350
PROJECTOR
PROPAGATOR FOR E
355
347
PROJECTOR
PROPAGATOR FOR A AND H
375
367
HIGH-LEVEL PROGRAMMING ELMENT(S)
120
FIRST STAGE WOVEN CODE BLOCK (INCORPORATING SIGNIFICANCES A, B AND C)
320
SECOND STAGE WOVEN CODE BLOCK (INCORPORATING SIGNIFICANCES D, E AND F)
340
THIRD STAGE WOVEN CODE BLOCK (INCORPORATING SIGNIFICANCES A, G AND H)
360

FIG. 4

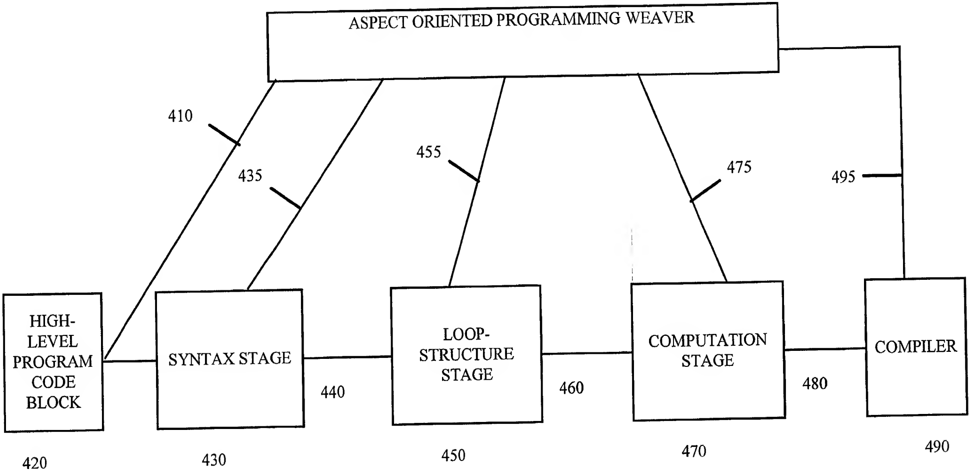
ASPECT ORIENTED PROGRAMMING WEAVER
410
435
455
475
495
HIGH-LEVEL PROGRAM CODE BLOCK
SYNTAX STAGE
440
LOOP-STRUCTURE STAGE
460
COMPUTATION STAGE
480
COMPILER
420
430
450
470
490

FIG. 5